# 1. DQL

DQL就是数据查询语言，DQL很常用，在以后完成的业务系统中，大部分功能都是和数据查询相关的，因此这是重点内容。DQL查询数据，返回结果集，DQL要使用的关键字是“SELECT”。

## 1.1 简单查询

“SELECT”最简单的用法是“SELECT 列名 FROM 表名”。例如：

|  |
| --- |
| *-- 查询表中所有数据* **SELECT** *\** **FROM** student; *-- 查询表中所有所有数据，这时指定了显示的列* **SELECT sid**, **name FROM** student; |

在列名中， 可使用“AS”关键字给该列设置别名，这样结果集的该列名称就是这个别名，其中“AS”关键字是可省略的，例如：

|  |
| --- |
| *-- 下面使用了一个常量列，该列名称是“专业”，值都是“计算机科学与技术”* **SELECT name**, **'计算机科学技术' AS '专业' FROM** student; *-- 也可将多个列的值合并成一个列，下面就是显示math、english和chinese的总成绩（相加得到）。适合于合并数值类型的字段。* **SELECT name**, (math + english + **chinese**) **AS '总成绩' from** student; *-- 当然可省略"AS"关键字*  **SELECT name '姓名' FROM** student; |

可以在列名前使用“DISTINCT”关键字，以去除此列结果中有重复的元素，例如：

|  |
| --- |
| **SELECT DISTINCT age FROM** student; |

会显示学生中所有的年龄。

## 1.2条件查询

DQL中可以使用“WHERE”关键字来附加查询条件，可使用的逻辑符号有：等于“=”（不同于类C编程语言，这里直接使用“=”表示等于，而不使用“==”）、大于“>”、小于“<”、大于等于“>=”、小于等于“<=”和不等于“!=”（SQL中，不等于也可使用符号“<>”）。

WHERE中如果有多个逻辑条件，则用关键字“AND”和“OR”连接，“AND”表示逻辑与，“OR”表示逻辑或。例如：

|  |
| --- |
| *-- 查询姓名不等于'张三'的学生* **SELECT** *\** **FROM** student **WHERE** name<> **'张三'**; *-- 可在where中使用and或者or来加上附加条件* **SELECT** *\** **FROM** student **WHERE** math > 90 **AND** english > 60; *# 查询三门总分大于200的学生* **SELECT** *\** **FROM** student **WHERE** (chinese+english+math)>200; |

可在WHERE中使用“列名 BETWEEN ... AND”关键字，表示对该列限定某段范围查询，例如：

|  |
| --- |
| *-- 查询数学成绩在1-10（包括）之间，并且英语成绩小于60的学生* **SELECT** *\** **FROM** student **WHERE** math **BETWEEN** 1 **AND** 10 **AND** english < 60; |

还可在WHERE子句中使用“IN”关键字，用法是“列名 IN(数值列表)”，表示只查询该列满足指定的数据列表中的值的数据。例如：

|  |
| --- |
| *# 查询数学分数为89, 90和91的学生* **SELECT** *\** **FROM** student **WHERE** math **IN** (89, 90, 91); |

## 1.3 为空或空字符串判断

有时需要判断数据的某列是否为空，数据为空是NULL，它和空字符串不同，空字符串是''。判空时，应该使用“IS NULL”（为空）和“IS NOT NULL”（不为空）。例如：

|  |
| --- |
| *# 查询年龄数据不为NULL的学生* **SELECT** *\** **FROM** student **WHERE** age **IS NOT NULL**; |

注意判断是否为NULL千万不能使用类似“age = NULL”和“age != NULL”这样的用法，因为标准SQL中规定，“=NULL”和“!=NULL”这样的写法都将返回0行结果集（即查不到数据）。

而要判断是否是空字符串，直接使用“= ''”和“!=''”是没有问题的。例如使用“AND”来查询年龄不为NULL且不为空字符串的学生：

|  |
| --- |
| **SELECT** *\** **FROM** student **WHERE** age **IS NOT NULL AND** age != **''** |

## 1.4 模糊查询

使用LIKE关键字进行模糊查询。

%表示匹配任意多个字符，\_表示匹配任意一个字符。比如：

|  |
| --- |
| *-- 比如查询姓张的学生* **SELECT** *\** **FROM** student **WHERE** name **LIKE '张%'**; *-- 查询姓李的，并且名字是两个字的学生* **SELECT** *\** **FROM** student **WHERE** name **LIKE '李\_'**; |

## 1.5 聚合查询

聚合函数有max()、min()、avg()、sum()和count()五个，它们分别用来计算某列数据的最大值、最小值、平均值、总和和个数。除了count()外，其他函数均适用于处理数值类型的列。例如：

|  |
| --- |
| *-- 查询学生的数学总分* **SELECT** *sum*(math) **AS '数学总分' FROM** student; *-- 查询数学平均分* **SELECT** *avg*(math) **'数学平均分' FROM** student; *-- count() 会扫描一列，返回该列不为空(NULL)的行数* **SELECT** *count*(**name**) **FROM** student; *-- 常用count(\*)来统计一张表有多少行，因为count(\*)会返回表中最大的行数。例如统计有多少学生：* **SELECT** *count*(*\**) **FROM** student; *-- 使用count(\*)和使用count(表达式)的效果是一样的，比如count(1)、count(2)和count('any')等。只要表达式不为null，就会返回整张表最大的行数。当然，若使用count(null)，结果总是0。* **SELECT** *count*(1) **FROM** student; **SELECT** *count*(**NULL**) **FROM** student; *-- 说明：为了统计表数据的行数，有人直接用count(\*)，有人喜欢用count(主键)，有人认为count(主键)效率比count(\*)高，说是因为count(\*)会扫描整张表的列统计行数，而count(主键)只会扫描主键列来统计行数。但据我在网上查找资料显示，并没有确切的说法，趋向于它们并无效率高低。开发中均可使用。* |

关于AVG函数：

|  |
| --- |
| *-- 统计总分大于250的人数* **SELECT** *COUNT*(*\**) **FROM** student **WHERE** (chinese+english+math)>250; *-- 统计语文成绩平均分* **SELECT** *SUM*(chinese)/*COUNT*(*\**) **FROM** student; *-- 当然可直接用AVG()函数。求总平均分：* **SELECT** *AVG*(chinese+english+math) **FROM** student; *-- 注意的是，如果该列为NULL，AVG函数是不会将它列入平均分的计算的，比较公平。 -- 即使上述是三列相加，如果其中一列是NULL，那么他们相加的结果还是NULL，因为NULL就是“未知”* |

## 1.6 分页查询

在网页中，当数据量大时，常用到分页，因此在数据库端就要做到只查询该页的数据，不能一次性查询到全部的数据返回给程序，这样是“假”分页，“假”分页并不能减轻数据库的负载。

MySQL中使用LIMIT关键字实现分页，LIMIT后面跟两个“参数”：一个是起始行，一个是查询几行，起始行从索引0开始。例子：

|  |
| --- |
| *-- 查询第一页数据，每页显示两条* **SELECT** *\** **FROM** student **LIMIT** 0, 2; *-- 查询第二页数据，每页显示两条* **SELECT** *\** **FROM** student **LIMIT** 2, 2; |

因此，分页查询当前页的SQL语句“公式”为：

|  |
| --- |
| SELECT \* FROM 表名 LIMIT (当前页 - 1) \* 每页显示条数, 每页显示条数 |

## 1.7 查询排序

使用“order by 列”指定按照哪个列排序，用asc和desc表示是升序（默认）还是降序。例子：

|  |
| --- |
| *-- 按姓名升序排列学生。asc是默认的，可省略。* **select** *\** **from** student **order by sname asc**; *-- 多个条件排序。先按照math成绩升序排序，若有math成绩相同的数据，再按照chinese成绩降序排序* **select** *\** **from** student **order by** math **asc**, chinese **desc**;  *-- 还可针对别名进行排序，使用更方便。例如： -- 按照总分从高到低排序：* **SELECT name '姓名'** chinese+english+math **'总分' FROM** student **ORDER BY '总分' DESC**; |

## 1.8 分组查询

分组查询使用“group by 列名”，将会把相同列名的数据分到一组，然后能获取组的信息。除了本列信息外，其他信息需要使用聚合函数。例如：

|  |
| --- |
| *-- 查询该表中男生和女生各有多少人* **select** sgender, *count*(*\**) **as '人数' from** student **group by** sgender; |

有时，需要在分组后再进行筛选，这时不能用where，而应该使用having来添加分组后的筛选条件。当然，分组之前需要使用条件时，还是使用where。

例如：

|  |
| --- |
| *-- 查询出至少有两个人的性别* **select** sgender **from** student **group by** sgender **having** *count*(*\**) >= 2; *-- 查询出大于20岁人中，至少有两个人的性别* **select** sgender **from** student **where sage** > 20 **group by** sgender **having** *count*(*\**) >= 2; |

# 2. 数据库的设计

## 2.1 数据的约束

数据库中需要对数据的存储进行约束，比如某字段不能为空，某字段值必须在某范围内等。数据库中进行约束主要有以下几种：默认值、非空、唯一、主键和外键。

### 2.1.1 默认值

创建表时可设置某字段的默认值，那么插入数据的时候，不过对此列不插入值，则此列就会用默认值。

|  |
| --- |
| **create table** student(  **sid int**,  **sname varchar**(20),  **saddress varchar**(50) **default '江苏'** *-- 设置默认地址为江苏* ); *-- 插入时若不插入saddress字段，默认就为江苏* **insert into** student(**sid**, **sname**) **values**(1, **'张三'**); |

对默认值字段插入null是可以的，并不是不允许此列为空。

### 2.1.2 非空

设置某字段非空，就是限制该字段必须赋值，不能为NULL。比如学生的学号不能为空。

|  |
| --- |
| **create table** student(  **sid int not null**, *-- 学号不能为空，插入时必须填写* **sname varchar**(20) ); *-- 执行此句会出错，因为sid必须插入。* **insert into** student(**sname**) **values**(**'张三'**); |

### 2.1.3 唯一

设置唯一就是该字段的值不能在本表其他条数据的该列中再出现，即不能重复。

|  |
| --- |
| **create table** student(  **sid int unique**, *-- 学号不能重复* **sname varchar**(20) ); **insert into** student(**sid**) **values**(1); *-- 再执行此句就出错，因为不能重复* **insert into** student(**sid**) **values**(1); |

上述中，执行多次：insert into student(sid) values(NULL)；是可以的，因为空值本身也不知道自己到底是什么值，因此可插入多次，并且唯一约并没有要求非空。

### 2.1.4 主键

主键约束就是要求此字段值非空且唯一。每个表只能有一个主键，用主键来唯一确定表中的一条数据。所以设计表时都要求有主键。

|  |
| --- |
| **create table** student(  **id int primary key**, *-- 设置sid为主键* **name varchar**(20) ); |

有时，可设置主键是“自增”的，即该列值是自动增长的，这样就不用手动维护表的主键。

|  |
| --- |
| **create table** student(  **id int primary key auto\_increment**, *-- 设置sid为主键自增* **name varchar**(20) ); *-- 插入时无需插入主键字段* **insert into** student(**name**) **values**(**'张三'**); |

实际上其他列也能使用自动增长，但要求该列必须是唯一的（unique）。同样，设置了自动增长列，插入时就可省略，但此时由于并不是每个字段都插入值，所以表名后要声明插入的字段，而不能省略，否则数据库无法知道数据是如何对应到各个字段的，例如上述的插入语句就不能这样写：

|  |
| --- |
| **insert into** student **values**(**'张三'**); -- 换成这样就是错误的 |

若实在不想写表名后的列名，又想自增的列是自动维护插入的，就可这样写：

|  |
| --- |
| **insert into** student **values**(NULL, **'张三'**); -- 换成这样就是错误的 |

即把自增的列值写为“NULL”，这样就会自增列也会自动维护自增。当然，自增列也可手动设置值：

|  |
| --- |
| **insert into** student **values**(12, **'张三'**); -- 换成这样就是错误的 |

那么下次插入记录，如果按照自增的话，id值会从12开始自增，即是13。

有两种方式选定表的主键。

一种是直接使用业务主键，例如使用学生的学号作为主键，该字段既是业务所需要的字段，也是能用于唯一标识一条记录的字段，因此可使用这样的字段作为主，这就叫业务主键。

第二种是使用逻辑主键，比如给表增加一个没有实际业务意义的字段（例如名为id的字段），该字段仅仅用于唯一标识一条记录，且选用它作为表的主键。我们所推荐的是这种方式，这样的方式完全保证了每条记录的唯一性，防止有时业务上的主键含义发生变化导致业务主键不能再成为主键字段。

定义主键有三种方式：

（1）方式一，写在创建表SQL的字段定义之后，同之前的示例，例如：

|  |
| --- |
| **CREATE TABLE** `student` (  **`id` INT PRIMARY KEY**,  **`name` VARCHAR**(100) ) |

（2）方式二，写在创建表SQL的所有字段定义之后，例如：

|  |
| --- |
| **CREATE TABLE** `student` (  **`id` INT**,  **`name` VARCHAR**(100),  **PRIMARY KEY**(**`id`**) ) |

（3）方式三，写在创建表的语句之后，即使用新的语句（ALTER TABLE）来修改表结构，例如：

|  |
| --- |
| **CREATE TABLE** `student` (  **`id` INT**,  **`name` VARCHAR**(100) ); *# 设置主键* **ALTER TABLE** `student` **ADD PRIMARY KEY** (**`id`**); |

上面语句可同时运行，在图形工具中，最好使用分号和换行作为每条SQL的结尾，更为清晰（实际上分号是可省略的）。

一般，推荐使用第三种方式设置表的主键，因为该方式可以把表先全部创建好，最后统一添加主键（以后外键等其他约束也能这样设置），好处是在创建表是由于没有约束，创建表的先后顺序没有影响（特别是外键约束，如果在创表时添加外键约束，则表的创建顺序是需要注意的，否则无法创建成功），且统一添加不容易遗漏。

也可以把表中的多个字段共同组合成一个主键，这样的主键叫联合主键。例如可以把姓名和年龄作为一个联合主键，这样就不允许插入姓名和年龄都相同的两个人。创建表的SQL可如下：

|  |
| --- |
| **CREATE TABLE** `student` (  **`name` VARCHAR**(100),  **`age` INT** ); *# 设置主键* **ALTER TABLE** `student` **ADD PRIMARY KEY** (**`name`**, **`age`**); |

### 2.1.5 外键

使用外键可约束表之间的数据，可解决数据冗余问题和不一致等问题。

例如： 员工表 和 部门表。当插入员工，填写员工所在部门的编号时，此编号必须是部门表的主键。只有这样才能使数据合理。

|  |
| --- |
| *-- 部门表* **create table** dept(  **did int primary key**,  **deptName varchar**(20) ); *-- 员工表* **create table** employee(  **eid int primary key**,  **empName varchar**(20),  **deptId int**, *-- 这是对应的部门id，下面就是对deptId做外键约束* **constraint** emp\_dept\_fk **foreign key**(**deptId**) **references** dept(**did**) ); */\* 声明外键约束的解释： emp\_dept\_fk是给此外键约束起个名字，一般以fk结尾表示外键 foreign key(本表中的外键列)； references表示参照，后面写上dept表和参照的did字段。表示参照此字段。  \*/* |

说明：被外键约束的表称为副表（或从表），约束别人的表称为主表，外键设置在副表上。

添加数据时，应先添加主表数据，再添加副表数据；当修改或删除数据时，应先改动副表数据，再改动主表数据。

前面提过，在写法上，还能在创建表之后通过ALTER TABLE语句来设置表的外键，语法是：

|  |
| --- |
| ALTER TABLE 表名 ADD CONSTRAINT 外键名称 FOREIGN KEY(外键字段名) REFERENCES 外键表名(对应的表的主键字段名); |

例如：

|  |
| --- |
| **ALTER TABLE** employee **ADD CONSTRAINT** emp\_dept\_fk **FOREIGN KEY**(deptId) **REFERENCES** dept(did); |

### 2.1.6 级联操作

添加了外键约束后，操作主表中数据时可能会受到制约。比如编号为EP001的员工在部门DP002中，那么此时删除部门DP002就会出错，因为有员工会“引用”它。

为了方便，有时希望可直接修改或删除主表中被外键关联的主键，那么可以使用级联操作。需谨慎使用。

若要使用级联修改，则使用关键字“on update cascade”，当主表的主键被修改时，副表相关的外键也会被修改；

若要使用级联删除，则使用关键字“on delete cascade”，当主表的主键被修改时，副表相关的外键也会被修改。

级联操作应设置在外键声明上。

例子：

|  |
| --- |
| *-- 部门表* **create table** dept(  **did int primary key**,  **deptName varchar**(20) ); *-- 员工表* **create table** employee(  **eid int primary key**,  **empName varchar**(20),  **deptId int**,  **constraint** emp\_dept\_fk **foreign key**(**deptId**) **references** dept(**did**) **on update cascade on delete cascade** *-- 使用级联修改和级联删除* ); |

级联操作必须在外键的基础上使用。

## 2.2 数据完整性总结

数据完整性就是通过对数据的约束来提高数据的正确性。数据完整性主要包括以下三个方面（涵盖了上一节讲到的几个概念）：

（1）实体完整性：实体就是一条记录，实体完整性约束表的一行是唯一的实体。通过定义主键约束来实现实体完整性。

（2）域完整性：域完整性指的是数据库表的列需要满足约束。约束的方面有：数据类型、长度、非空约束、唯一约束等。

（3）参照完整性：参照完整性是对于多表来说的，表现为外键约束。参照完整性可避免数据冗余和数据不一致。

## 2.3 数据库设计的范式

设计数据库表时应尽量遵守范式，这样使得数据库的结构更好。

（1）第一范式：表的每个字段必须是不可分割的独立单元。比如在name字段中只存储人的真实姓名，不应存储人的昵称等信息。若需要存其他信息，应该新建字段存储。

（2）第二范式：在第一范式的基础上，要求每张表只表达一个意思。表的每个字段都和表的主键有依赖。

比如在员工表中，可存储员工的基本信息，如姓名、单位等信息，但是不能存储员工的订单信息。

（3）第三范式： 在第二范式基础上，要求每张表的主键之外的其他字段都只能和主键有直接决定依赖关系。

比如员工表：员工编号、姓名、部门编号、部门名。这就违反了第三范式（但符合第二范式），需要拆分表来降低数据冗余。可设计成员工表和部门表，即：

部门表：部门编号、部门名；员工表：员工编号、姓名、部门编号。

一般我们设计数据库都应该尽量遵守到第三范式，这意味着我们需要拆表。拆表后就涉及到表与表之间的关系。表之前有一对多、多对多和一对一的关系，其中都用到外键关联。读者可以自行研究（这里暂时不再细讲，可读《数据库系统概念》），当然这部分内容在以后学习Hibernate框架时也会提及。学习Hibernate时，其中能体会到对应的Java类时如何设计的。当然，在里面讲一对一关系时，有两种实现方式，但实际中，一对一的关系用的不多，因为一对一关系可以直接设计在一张表中，还能提高查询效率（除非信息非常多）。

# 3. 高级内容

## 3.1 关联查询（多表查询）

很多情况下，需要一次从多个表中查询到数据。比如上面讲到的部门表和员工表，我们需要查询该部门下所有的人员信息，或者查询一个员工所在的部门。

现在我们以查询所有员工和对应的部门名称为例。

（1）交叉连接查询。直接在多张表中查找数据。

|  |
| --- |
| **SELECT** deptName, empName **FROM** dept, employee; |

这种方式会查询到很多“错误”数据，因为直接选择两张表进行交叉连接，使用的是“笛卡尔乘积”。笛卡尔乘积也可使用“CROSS JOIN”作为连接关键字，例如可将上述SQL写为：

|  |
| --- |
| **SELECT** deptName, empName **FROM** dept **CROSS JOIN** employee; |

使用笛卡尔乘积产生的数据量会很大，效率低下。为了正确查询到需要的数据，要避免产生笛卡尔乘积，一般用以下步骤来进行多表查询：

* 确定需要查询哪些表，将这些表写在from后面；
* 确定要查询哪些字段，将这些字段写在select后面；
* 确定表与表之间的连接条件。连接条件数量是表的数量 - 1。比如若有三张表连接查询，则至少需要三张表之间的两个条件。条件写在where中。其余自定义的条件也写在where中。

（2）内连接查询。现按照上述步骤写查询语句。

|  |
| --- |
| **SELECT** deptName, empName *-- 2.确定查询的字段* **FROM** dept, employee *-- 1. 确定查询的表* **WHERE** dept.did = employee.eid; *-- 确定表连接的条件* |

这样查询就会显示正确的数据了。使用多表查询，经常在字段前面指明表名，比如“dept.did”，这样防止多张表中有相同名称的字段会产生歧义，使得SQL无法执行。在dept和employee中，他们之间无相同名称的字段，因此也可不指明表名。

有时表名很长，写表名会很麻烦，所以在from后的表名中，用as定义表的别名以便使用，比如：

|  |
| --- |
| **SELECT** d.deptName, e.empName **FROM** dept **as** d, employee **as** e *-- 定义dept表别名为d，employee表别名为e然后就能在select后和where后使用* **WHERE** d.did = e.eid; |

其中，as是可以省略的，直接在表名后写别名即可：

|  |
| --- |
| **SELECT** d.deptName, e.empName **FROM** dept d, employee e **WHERE** d.did = e.eid; |

内连接查询只会显示满足条件的数据，如果一个部门下没有员工，该部门也不会显示。

内连接还常用以下的写法：

|  |
| --- |
| **select** empName, deptName **from** employee **inner join** dept **on** eid = did; |

即在要连接的表之间写inner join，表的连接条件用on表示。当然，自定义的查询条件还是写在where中，比如（下面使用了别名）：

|  |
| --- |
| **SELECT** d.deptName, e.empName **FROM** dept d **INNER JOIN** employee e **ON** d.did = e.eid **where** e.empName = **'张三'**; |

（3）左（外）连接查询

左连接查询就是使用左边表的数据按条件连接右边表的数据，除了会显示全部匹配的结果，还会显示左表中未能匹配的结果，此时右表结果显示NULL。

即左连接就是会显示左表中所有的数据，若右表中午匹配的，会显示NULL。比如员工表，若一个部门下无员工，当部门表作为左表时，还是会显示此部门，只是员工显示为NULL，和内连接稍有区别。左连接使用left outer join，其中outer可省略。在left outer join左边的就是左表，右边的就是右表。

例如：

|  |
| --- |
| **select** d.deptName, e.empName **from** dept d **left join** employee e **on** e.eid = d.did; |

（4）右（外）连接查询

和左外连接类似，只不过总是显示右边表的数据，只是左右方向交换了而已，使用关键字right outer join，在此关键字右边的是右表。例子：

|  |
| --- |
| **select** d.deptName, e.empName **from** employee e **right outer join** dept d **on** e.eid = d.did; |

## 3.2 子查询

子查询就是嵌套查询，子查询的语句放到小括号之内。例如下述查询：

学生和教师是多对多关系，查询id为1的教师所教过的所有学员：

|  |
| --- |
| **SELECT** *\** **FROM** students **WHERE** id **IN** (**SELECT** s\_id **FROM** teacher\_student **WHERE** t\_id=1); |

## 3.3 存储过程

存储过程是具有逻辑处理的SQL语句。在存储过程中可以使用流程控制，如if、while等，和类C语言稍有区别。

使用存储过程注意下面两个特点：

（1）存储过程是在数据库服务器端执行，速度快，效率高；

（2）不同数据库的存储过程使用方法有不同，移植性差。

下面以简单的案例介绍存储过程的使用。

（1）简单的存储过程案例：

|  |
| --- |
| *-- 声明存储过程的结束符为$* **delimiter** $; *-- 下面定义一个名为proceTest的存储过程* **create procedure** *proceTest*() *-- 括号中可写参数列表，此例中没有* **begin** *-- 以begin表示开始存储过程  -- begin和end之间就写要执行的语句即可* **select** *\** **from** student; *-- 执行一个简单的查询* **end** $; *-- 用end和结束符表示此存储过程定义的结束  -- 最后使用call来调用存储过程* **call** *proceTest*(); *-- 可使用drop删除存储过程* **drop procedure** *proceTest*; |

（2）带有参数的存储过程。

存储过程的参数也是写在存储过程名的小括号中，参数可标记为in、out或者inout。

in表示此参数是输入参数，参数数据可传递到存储过程中使用；

out表示此参数是输出参数，该参数用于从存储过程中返回结果；

inout表示此参数是输入输出参数，既可以输入，也可以输出。

例1：带有输入参数的存储过程。

|  |
| --- |
| **delimiter** $; **create procedure** *proTest*(**in** empId **int**) *-- 参数格式：in/out/inout 参数名 参数类型* **begin  select** *\** **from** employee **where** eid = empId; **END** $;  *-- 调用* **call** *proTest*(1); |

在使用输出参数之前，要介绍下MySQL中的变量。

MySQL中，已预先设置了很多变量，这些是全局变量，可以使用语句“show variables”查看所有的全局变量。

若想查看某个全局变量，使用“select @@全局变量名”，比如“select @@port”查看端口。再介绍两个全局变量：

character\_set\_client变量值表示MySQL服务器接收数据的编码；

character\_set\_results变量值表示MySQL服务器输出数据的编码；

可使用set命令给变量赋值，比如“set @@character\_set\_client=’gbk’”。

除了全局变量，用户可自己定义变量。用户自定义的变量有“会话变量”和“局部变量”之分。会话变量只存在于当前客户端与数据库的一次连接当中，如果连接断开，则会话变量就会丢失。而局部变量是仅仅在存储过程中定义和使用的变量，当存储过程执行完毕，局部变量就丢失。

为了区分，定义和使用会话变量是在变量前加上一个@，而局部变量什么也不加，查看和设置变量值还是使用select和set命令。

例2：带有输出参数的存储过程。

|  |
| --- |
| **delimiter** $; **create procedure** *proTest*(**out** str **varchar**(20)) **begin** *-- 给参数赋值。在这里，str是局部变量。* **set** str = **'hello world'**; **END** $;  *-- 用一个会话变量来接收存储过程输出的str参数。* **call** *proTest*(*@myStr*); *-- 直接用@变量即可“定义”并“使用”。 -- 显示得到值的会话变量* **select** @myStr; |

例3：带有输入输出参数的存储过程。

|  |
| --- |
| **delimiter** $; **create procedure** *proTest*(**inout** n **int**) **begin** *-- 先查看变量* **select** n;  *-- 再设置值* **set** n = 500; **END** $;  *-- 设置一个会话变量值* **set** *@num* = 10; **call** *proTest*(@num); *-- 调用* **select** @num; *-- 显示结果* |

（3）带有流程控制的存储过程

例1：创建一个存储过程，若传递整数1，则返回“星期一”；若为2，则返回“星期二”，若为3，则返回“星期三”，否则返回“输入错误”。

|  |
| --- |
| **delimiter** $; **create procedure** *testIf*(**in** num **int**, **out** str **varchar**(20)) **begin  if** num = 1 **then  set** str = **'星期一'**;  **elseif** num = 2 **then  set** str = **'星期二'**;  **elseif** num = 3 **then  set** str = **'星期三'**;  **else  set** str = **'输入错误'**;  **end if**; **end** $;  **call** *testIf*(4, *@str*); **select** @str; **call** *testIf*(2, @str); **select** @str; |

例2：输入一个整数，求1加到此整数的和。

|  |
| --- |
| **delimiter** $; **create procedure** *testWhile*(**in** num **int**, **out** result **int**) **begin** *-- 使用局部变量时，要先定义，默认值为1* **declare** i **int default** 1;  *-- 设置总和result开始为0* **set** result = 0;  **while** i <= num  **do  set** result = result + i; *-- SQL中没有+=的用法* **set** i = i + 1;  **end while**; **end** $;  **call** *testWhile*(100, *@sum*); **select** @sum; |

显示结果为5050。

（4）将查询的结果给out变量带回输出，使用into。

例子：

|  |
| --- |
| **delimiter** $; **create procedure** *findById*(**in** id **int**, **out** eName **varchar**(20)) **begin   select** empName **into** eName **from** employee **where** eid = id; **end** $;  **call** *findById*(1, *@name*); **select** @**name**; |

此方式只适用于select的结果只有一行结果集。其他的情况需要自己学习。

## 3.4 触发器

当操作某操作时，希望同时触发执行一些动作，可以使用触发器。例如：当向员工表插入一条记录时，希望同时往日志表中插入数据。

例子（需要自行创建日志表test\_log，里面有content字段）：

|  |
| --- |
| *-- 日志表* **CREATE TABLE** test\_log (  **content VARCHAR**(100) );  *-- 插入员工表时的触发器：* **CREATE TRIGGER** trigger\_empAdd **AFTER INSERT ON** employee **FOR EACH ROW  INSERT INTO** test\_log (**content**) **VALUES** (**'员工表插入了一条记录'**); *-- 更新员工信息时的触发器* **CREATE TRIGGER** trigger\_empUpd **AFTER UPDATE ON** employee **FOR EACH ROW  INSERT INTO** test\_log (**content**) **VALUES** (**'员工表更新了一条记录'**); *-- 删除员工信息时的触发器* **CREATE TRIGGER** trigger\_empDelete **AFTER DELETE ON** employee **FOR EACH ROW  INSERT INTO** test\_log (**content**) **VALUES** (**'员工表删除了一条记录'**); |

## 3.5 MySQL权限管理

MySQL的root账户拥有所有权限。实际中，root用户不能随意使用，有时需要进行用户权限管理，比如新建一个用户，并给该用户分配适当的操作权限。

（1）修改用户密码

用户信息保存在mysql数据库的user表中，执行：

|  |
| --- |
| **use** mysql; **select** *\** **from user**; |

可看到账户表。其中可看到“authentication\_string”就是用于保存密码的，但是这个密码是用MD5加密后显示的字符串。

那如何修改密码呢？我们只要用自定义密码的MD5形式替换当前的值即可。MySQL提供了password()函数来计算一个字符串的MD5值，比如执行“select password('love')”得到的就是love的MD5字符串。可用此函数来加密用户的密码信息。

因此这里可执行以下语句修改密码：

|  |
| --- |
| *-- 修改密码* **update user set authentication\_string** = *password*(**'我的密码'**) **where user** = **'要修改的用户名'**; *-- 刷新权限使生效* **flush PRIVILEGES** ; |

这样密码就会被修改。如果你不执行flush PRIVILEGES;的话，那么必须重启MySQL服务，新密码才会生效。

其他的权限管理自行学习。

## 3.6 MySQL的备份和恢复

这里讲如何备份和恢复MySQL中的指定数据库。

一、备份数据库

使用备份命令：

|  |
| --- |
| mysqldump -h 主机名 -u 用户名 -p 密码 要备份的数据库名 > 备份保存的文件 |

例如：

|  |
| --- |
| mysqldump -h localhost -u root -p 123456 mytest > d:/mytest.sql |

这样就会备份mytest数据库到mytest.sql文件中。这里使用的路径形式不是Windows中的反斜杠，也是能识别的。

二、恢复数据库

方式一：在MySQL命令行中，创建并选择这个数据库（例如mytest数据库），并执行：

|  |
| --- |
| SOURCE 备份的文件 |

例如这里就执行“SOURCE d:/mytest.sql”命令，这样数据库就会被恢复。

方式二：直接执行命令：

|  |
| --- |
| mysql -u 用户名 -p 密码 数据库名 < 备份的文件 |

例如：mysql -u root -p123456 mytest < d:mytest.sql。